

A Novel Approach to Implement Message Level Security in RESTful Web Services

Gyan Prakash Tiwary^a, Abhishek Srivastava^b

*Discipline of Computer Science and Engineering
Indian Institute of Technology Indore
Indore, India 452020*

^aEmail: phd1501101003@iiti.ac

^bEmail: asrivastava@iiti.ac.in

Abstract

The world is rapidly adopting RESTful web services for most of its tasks. The once popular SOAP-based web services are fast losing ground owing to this. RESTful web services are light weight services without strict message formats. RESTful web services, unlike SOAP, are capable of message transfer in any format be it XML, JSON, plain-text. However, in spite of these positives, ensuring message level security in REST is a challenge. Security in RESTful web services is still largely dependent upon transport layer security. There has been some work recently towards message level security in such environments wherein the transfer of message level security metadata is done through utilising new HTTP headers. We feel, however, that any method that compromises the generality of the HTTP protocol should be avoided. In this paper, therefore, we propose two new ways of encryption that promise to ensure message level security in RESTful web services without the need for special HTTP headers. This approach works seamlessly on most famous content-types of RESTful web services: XML, JSON, HTML, plain-text and various ASCII printable content types. Further, the proposed approach removes the need for content negotiation in cases where the content comprises XML, JSON, HTML, plain-text, and ASCII printable content types and also removes the need for XML or JSON canonicalization.

Keywords: RESTful Web Services, Message level Security

1. Introduction

Web services are a means to access the web in a ‘programmatic’ manner. A website and a web service are similar in that both respond to requests made by clients on the web. Websites respond with content that can easily be comprehended by a human eg. HTML, CSS and Javascript, whereas web services respond with content meant for consumption by other applications eg. XML, JSON. The response from web services has a greater focus on data whereas that from websites is more towards an interactive representation of data.

Web services may be categorised into two broad types: SOAP-based web services and RESTful web services. SOAP-based web services deal with properly structured messages comprising formal XML-based formats for a request and response. RESTful web services, on the other hand, lack such formal formats. This contributes to the flexibility and ‘light weight’ nature of RESTful web services. SOAP request and response messages usually utilise HTTP or SMTP packets as containers, whereas REST requests comprise a simple HTTP request with the use of the common HTTP verbs CRUD (create, read, update and delete) for executing operations on the resource [1]. The response from a RESTful web service to a request is also an HTTP response containing XML, JSON, CSV, HTML, or plain-text.

In REST, any content or service on the web for which a client makes a request to a server is known as a ‘resource’. A resource is content available with the server side and may get transferred

or even modified based on a client’s request. A resource may be a text or a binary file or may be data stored in a database. Each resource is identified by a unique ID called the Uniform Resource Identifier (URI). A resource may be represented in various formats such as XML, JSON, CVS, HTML, plain-text and others with the most popular representation today being XML and JSON.

RESTful web services, unlike SOAP, do not have a formal description language, therefore, it is common for services to publish their resource representations on public domains like websites. For example, Google publishes the resource representation of its drive API at the following URL <https://developers.google.com/drive/v3/reference/about#resource>. In addition to this, web service providers also describe the capabilities of the various HTTP verbs for performing operations on a given resource in a similar manner.

An important requirement for web services while connecting with their clients is a robust security mechanism. This is along lines similar to ordinary websites. Security is imperative for the following three purposes: confidentiality, integrity and authenticity. Confidentiality implies that a conversation between the server and the client makes no sense to a potential intruder eavesdropping on the conversation. Confidentiality can be provided by encryption of the message. Integrity implies that a message in transit between a client and a server must remain unchanged during transit. Authenticity ensures that a client and server talking to each other are indeed talking to each other and

not a third entity. Digital signature provides integrity and authenticity.

Web services and websites both work at the application layer. To use a website or web service a client first needs to connect with the server providing the service at the transport layer. This type of connection is called a TCP (Transmission Control Protocol) connection. TCP contains and carries the application layer data in the form of HTTP or SMTP inside it. There are quite a few Security mechanisms available at the transport layer that are quite robust and provide encryption, authentication and authorization. One such mechanism is Transport Layer Security (TLS) [2].

TLS makes use of two types of encryption algorithms: symmetric key encryption and asymmetric key encryption algorithms. In asymmetric key encryption, the server maintains two keys of which one is 'public' and the other 'private'. The server makes the public key available to all, and the private key is kept hidden. A message that is encrypted using the public key can only be decrypted using the private key. The client encrypts the message using this public key and sends it to the server. The server subsequently decrypts the message using the secret private key. In symmetric key encryption, both the server and client maintain a shared secret key between them called the symmetric session key. The symmetric session key is used both for encryption and decryption. To send a message, the client encrypts the message with this key and upon reaching the server, the same key is used for decryption.

To transfer a large amount of data symmetric key algorithms are efficient as these are relatively light weight. To transfer fewer data asymmetric key algorithms are deemed more appropriate. The big question with symmetric key encryption is: how to transfer the symmetric key initially from the client to server over a non-secure channel? A server provides its public key to all its clients. A client that wants to connect to the server provides the symmetric shared key information encrypted with the server's public key to the server. The server sees this shared key information after decrypting the client's message using the private key. After the server's agreement on the shared key both start to transfer their data encrypted by the shared symmetric key. This is the normal sequence followed in encryption: first, the asymmetric key encryption, as described earlier, is used only to securely get the symmetric key for a session. Subsequent to this, in all further communications in that session between client and server symmetric key encryption is used.

Transport Layer Security (TLS), as described above, is normally used to encrypt the TCP container that contains the HTTP or SMTP container (application layer data) of the web service request and response. We know that both SOAP and RESTful web services contain XML, JSON etc inside the application layer packet (HTTP or SMTP), which is contained in the transport layer packet (TCP). TLS is effective with web services if the transfer of messages is between two parties only. In scenarios where several intermediate nodes need to access different parts of the content of the same HTTP or SMTP container, however, TLS becomes ineffective and the need arises for a message level security mechanism at the application layer [3]. This is a typical requirement of a web service composition scenario,

where several nodes interact with each other to provide a larger composite application.

For better comprehension of the scenario, let us consider a simple real world example of a big box containing a small box inside which there are several tennis balls with some relevant information written on them. Each ball belongs to a different person. The big box here can be looked upon as the HTTP container, the small box represents the entire XML or JSON message, and the balls represents various tags in XML or fields in JSON. The message written on one ball should be such that it cannot be understood by a person to whom it does not belong. What TLS does is: it encrypts the big box (Fig. 1.). When a person decrypts this big box, the small box and the information written on all the balls inside the small box become visible to that person irrespective of whether the ball belongs to him or not. Message level security, on the other hand, encrypts the information written on each ball and therefore only the person to whom a ball belongs is given the rights to decrypt the message. The balls here are the different tags of XML (or fields in JSON) that may be intended for different intermediate web service nodes in a web-service composition scenario. The entire XML should be encrypted so that the intermediate nodes are only able to understand and/or edit tags that are meant for them. The need, therefore, is to encrypt the message written on each tag separately with different keys (Fig. 2.). Encrypting the various parts of an XML or JSON document with different keys is known as message level encryption. Message level encryption has the potential to provide message level security to web services.

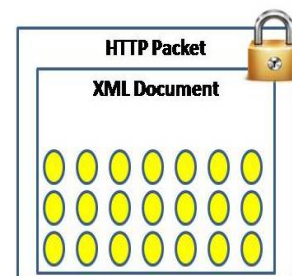


Figure 1: Encryption by TLS

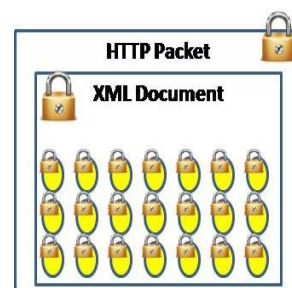


Figure 2: Message Level Encryption

SOAP based services do have formal mechanisms for message level security defined by W3C and OASIS [3] which pro-

vide authentication, authorization and encryption. These services are broadly based on XML Signature [4] and XML Encryption [5]. XML signature and XML encryption may work on the whole document, a part of the document, or even on binary documents. After encryption and signature, however, the encrypted document gets represented as XML only.

XML encryption uses existing block or stream ciphers to encrypt the document. Some of these algorithms are Advanced Encryption Algorithm (AES) [6], Data Encryption Algorithm [7] etc. Both XML encryption and signature use different versions of the Standard Hash Algorithm [8] for creating the message digest. For key agreement they use the Elliptic Curve Diffie-Hellman (ECDH) [9] Key Agreement. All these encryption and hash algorithms work on the data at the byte level.

The issue with byte level encryption and signature is that two XML documents that represent the same data may not be identical byte-by-byte and as such even a slight cosmetic difference between the documents changes the whole encryption or signature. An example of this is: interchanging the position of attributes in a tag does not change the nature of the XML document, however, it completely changes the encryption and signature of that document. The normal procedure to overcome this problem is through XML canonicalization [10]. Here, several strict rules need to be applied on the XML documents such that two XML documents that represent the same data become identical down to the last byte. Effective implementation of XML based signature and encryption requires the compulsory step of XML canonicalization.

This technique is ineffective if the two parties in conversation support different technologies, e.g. one supports XML content and the other supports JSON. Also, XML canonicalization is a compulsory step in XML based security and it is a cumbersome process indeed. All this contributes to making such security techniques unsuitable for RESTful web services. Currently, there is no formal model for the provision of message level security in RESTful web services. In RESTful web services, both message encryption and authentic connection are still majorly dependent upon Transport Layer Security (TLS).

Web service composition is a typical example where intermediate nodes need to access the same RESTful message and where TLS is not effective. In web service composition several service providers work together to form a large composite service offering. Each web service in a service composition does its bit and the larger composite task gets done. We are also moving towards a scenario where individual web services dynamically partake in a composition and after providing their respective services exit with the group. To realise such a scenario in a secure environment, it is imperative that an effective message level security mechanism be in place that would enable a web service only partial access to a message and keep the rest of it hidden.

We need to keep the various REST principles in mind while designing message level security models for RESTful services. In recent work by Gabriel Serme et. al [11] use of HTTP headers is made to transfer names of encryption algorithms, keys and other security meta data. Encryption and signature in this approach are done through the use of existing algorithms. We

feel that sending new headers in HTTP packet makes the HTTP protocol itself non general. Further, use of existing encryption algorithms results in large sized HTTP contents.

In this paper, we propose a novel approach to encrypting RESTful web services at the message level that works well simultaneously with most popular content types (XML, JSON, HTML and plain-text). The main idea in this approach is to replace the ASCII printable characters with a series of numbers during encryption and doing the reverse during decryption. Encryption and decryption through this simple approach become easy and effective. This is because the encrypted message is quite small in size as compared to that of existing algorithms. In fact, in certain situations, the encrypted message is smaller than the message itself. The character to number conversion proposed also removes the need for XML canonicalization because it is not a byte-by-byte encryption.

The proposed approach also eliminates the need for content negotiation wherein the client and server negotiate on the type of content (XML, JSON) to be used for communication. This is because a resource gets converted into the same encryption irrespective of the type of representation. The series of numbers that constitutes the encryption will always be placed in the HTTP body as text/plain content type. The approach does not require special HTTP headers to pass security meta-data nor does it violate REST principles.

Resource Representation is all about one to one mapping of XML tag-name and data type supported inside that tag, or data type supported by the value of the JSON name. Several web service providers represent their resources in their API documentation. In the proposed approach the server does not need to publish its resource representation. This is because all representations (XML, JSON etc.) get converted to the same encryption and can be decrypted to any representation. Data types in between the tags and JSON names are specified explicitly in the encryption itself. The client comes to know about resource representation in the encrypted form after its first request for the resource.

It should be noted that in this paper we do not intend to provide a complete security solution for web service composition scenario. The idea is to introduce a novel, and much simpler technique for encryption that is also secure, reliable, fast, and results in much smaller sized encrypted messages. The technique also removes the need for XML canonicalization and content negotiation.

The paper is organized as follows: in Section 2, we discuss the system architecture for our approach. In Section 3 we describe our proposed approach in detail with a running example for better understanding. We discuss various advantages of our approach in Section 4 and conclude the paper in Section 5.

2. System Architecture

In RESTful communications and in general XML-tags, XML-attributes, values of XML-attributes and JSON-names change less frequently and therefore we call these the *non-variable* parts of the request and response messages. The values

in between XML-tags or the values of JSON names on the other hand change much more frequently and we call these the *variable* parts of request and response messages. For example:

XML 1:

```
<root attr1="value1" attr2="value2">
<name>iiti</name>
<value>2</value>
</root>
```

JSON 1:

```
"root": {
  "-attr1": "value1",
  "-attr2": "value2",
  "name": "iiti",
  "value": "2"
}
```

XML and JSON can be easily converted into each other. An XML and its corresponding JSON are shown above as XML 1 and JSON 1. In XML 1 the “root”, “name”, “value”, “attr1”, “attr2”, “value1” and “value2” are the *non-variable* parts and “iiti” and “2” are the *variable* parts of the message. In the corresponding JSON message JSON 1, we have similar *non-variable* and *variable* parts.

In our approach of encryption both the *non-variable* and *variable* parts of the message are represented by a string of numbers. The encrypted message is sent to the receiver encapsulated within an HTTP packet as plain-text. We know that XML and JSON can be easily converted to each other and as stated earlier the strength of this algorithm is that it does not matter whether the content to be encrypted is in JSON or XML the result is the same encrypted message which can subsequently be decrypted into JSON and/or XML as the need be. Encryption and decryption can be done by a shared symmetric key among parties. In case of web service composition where different intermediaries have access to the same RESTful message, different parts of the same message can be encrypted with different symmetric keys. We initially discuss message level encryption between two parties only. Subsequently, in Section 3 we escalate the discussion to multiple party web-service composition.

Figure 3 shows the various components at the client and server ends that are used for encryption/decryption. At the server side, there is a message level encryption/decryption module (Encryption-Decryption Engine) that lies between the RESTful service application and the HTTP service. At the client side, the same lies between the RESTful client and the HTTP client. There is a ‘Key Manager’ at both ends that is connected to the corresponding HTTP server and client.

2.1. Key Manager

Prior to the start of a conversation, both ends require a symmetric session key. The key manager at both ends manages the symmetric key. First, the client’s key manager sends a key request to the server’s key manager as a special command (“Get

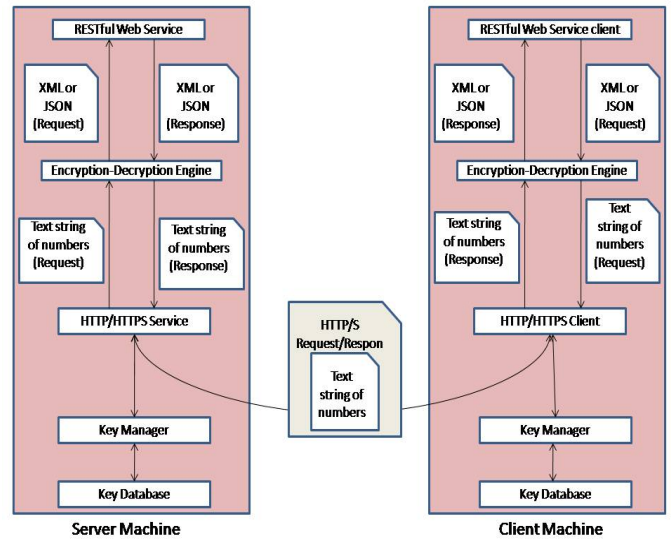


Figure 3: Client Server Interaction

key”). The “Get key” command goes to the HTTP client section at the client side. The HTTP client section at the client side places the “Get key” command in the body of the HTTP POST request with the content type being plain-text as shown in Figure 4.



Figure 4: Client’s request for key

The HTTP service at the server side receives the request and finds the “Get key” string in the POST request’s body. Subsequently, the server’s key manager generates 10 random integers within a given range and creates a unique *10 element key*. The server’s key manager passes this *10 element key* to the HTTP service at the server side. The HTTP service at the server side creates an HTTP response, places the key in the response body with the content type being plain text and sends the HTTP response to the HTTP client at the client side (Fig. 5). The HTTP client at the client side reads the response body and passes the content to the client’s key manager.

The conversation between the client and the server for a key is a matter between just two parties and therefore the key exchange can utilise TLS for security. Both sides store the shared key in a key database for further use. The number of keys maintained by each member in a web services composition group depends on the structure of the web services composition. At most, each member in a group of n service providers may need

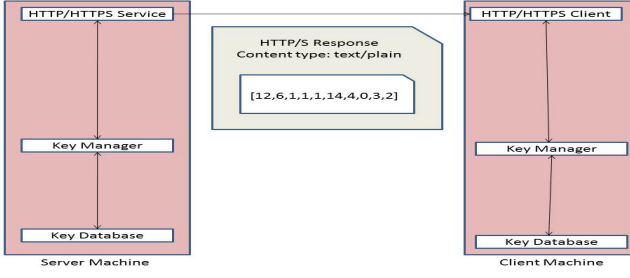


Figure 5: Server's response with key

	6	5	4	3	2	1
18	Q	1	2	3	4	5
17	6	7	8	9	A	B
16	C	D	E	F	G	H
15	I	J	K	L	M	N
14	O	P	Q	R	S	T
13	U	V	W	X	Y	Z
12	a	b	c	d	e	f
11	g	h	i	j	k	l
10	m	n	o	p	q	r
9	s	t	u	v	w	x
8	y	z				
7						

Figure 6: Temporary Table (TT)

to maintain n secret keys ($n-1$ keys to communicate with the other $n-1$ members and one group key). The shared key gets stored in key-databases of the client and the server. The client is responsible for a change in key. To change the key the client sends a new key request to the server.

2.2. 10 Element Key

This symmetric key is at the heart of the proposed approach and comprises 10 elements. Using just this key both the client and server can create various tables that take part in message level encryption and decryption. We have seen an example of the 10 element key randomly generated by the server in Figure 5 ($[12,6,1,1,1,14,4,0,3,2]$). In general, the key is represented as $key = [rows, cols, start_with, row_rev, col_rev, symbol_type, group_size, reverse, final_sum, power]$. In this paper different elements of the key are referred to by using indexing starting from 0, for example $key[0]$ for rows, $key[1]$ for cols etc. We will now understand the various elements of the key one by one.

2.2.1. rows ($key[0]$)

The Encryption-Decryption engine at both the client and server sides maintains a table that we choose to call the *Temporary Table (TT)*. rows at $key[0]$ specifies the number of rows in this table. The number of rows in $TT = key[0] + 1$. $key[0]$ ranges from 1 to any integer depending upon the capability of the system.

2.2.2. cols ($key[1]$)

cols at $key[1]$ specifies the number of columns in the TT. The number of columns in $TT = key[1] + 1$. For $key = [12,6,1,1,1,14,4,0,3,2]$ the TT at both the client and server sides will have 13 rows and 7 columns (including the coloured 'header' rows and columns) as shown in Figure 6. The coloured cells of the table are used to store table headers, whereas the white cells are for normal entries. $key[1]$ ranges from 1 to any integer depending upon the capability of the system.

2.2.3. start_with ($key[2]$)

We have two types of headers in TT, one is a row header (rh) and the other is the column header (ch). As part of the encryption process, we need to number both rh and ch. Numbering always starts with 1. Numbering can start from either rh or ch depending upon the value of $key[2]$. The header numbering of

TT starts with rh if the value of $key[2]$ is 0 and it starts with ch if the value of $key[2]$ is 1. Range of $key[2]$ is {0, 1}. If the numbering of the header starts with numbering rh, then the same count continues while numbering ch and vice-versa. For example, in $key = [12,6,1,1,1,14,4,0,3,2]$ the header numbering starts with ch. In Figure 6, therefore, the count 1 to 6 has been used for numbering ch and in continuation with that 7 to 18 have been used for numbering rh.

2.2.4. row_rev ($key[3]$)

If $key[3]$ is 0 then the rh numbering starts from the top most row (top to bottom) and if $key[3]$ is 1 then the rh numbering starts from the bottom row (bottom to top). For $key = [12,6,1,1,1,14,4,0,3,2]$ the rh numbering starts from the bottom and progresses towards the top as shown in Figure 6. The range of $key[3]$ is {0, 1}.

2.2.5. col_rev ($key[4]$)

If $key[4]$ is 0 then the ch numbering starts from the left most column (left to right) and if $key[4]$ is 1 then the ch numbering starts from the right most column (right to left). For $key = [12,6,1,1,1,14,4,0,3,2]$, the ch numbering starts from the right and progresses towards the left as shown in Figure 6. The range of $key[4]$ is {0, 1}.

2.2.6. symbol_type ($key[5]$)

There are four types of characters that can be used in this table as elements in the non-header cells: small alphabet, capital alphabet, digits, and special symbols. The total possible arrangements in this table are: ${}^4C_1 \times (1!) + {}^4C_2 \times (2!) + {}^4C_3 \times (3!) + {}^4C_4 \times (4!) = 64$. For all these 64 arrangements $key[5]$ ranges from 0 to 63. Each value defines a unique occurrence, non occurrence, and order of occurrence of the various printable ASCII characters in the non header cells of the TT. For each value of $key[5]$, the entry of characters in TT starts from the top left corner non-header cell and sequentially moves towards the bottom right corner non-header cell. For example, 1) If value $key[5]$ is 0 then only small alphabet characters are allowed in TT, 2) If it is 1 then only capital alphabet characters are allowed in TT. Similarly, if $key[5]$ is 14 then the order of entering characters in TT is digits, followed by capital alphabets, further followed by small alphabets. The value 14 ignores special symbols. There are various other arrangements of characters for other values of $key[5]$. Another value of $key[5]$ may

have included special symbols and the various characters may have been in a different order of occurrence. Figure 6 shows the entry of *TT* if $key = [12, 6, 1, 1, 1, 14, 4, 0, 3, 2]$. $key[5]$ and the corresponding arrangement of characters are only assumptions of the authors. During implementation, these may be different, but the total possible combination is always fixed.

2.2.7. *group_size* ($key[6]$)

Various groups of consecutive elements in *TT* are created. Each group has a size indicated by $key[6]$. The last group of *TT* can have fewer elements than $key[6]$. For $key = [12, 6, 1, 1, 1, 14, 4, 0, 3, 2]$ since $key[6]$ is 4, *TT* has various groups of 4 elements each. In Figure 6 we represent the various groups as alternately underlined and non-underlined elements. The elements 0, 1, 2, 3 are in the same group and 4, 5, 6, 7 are in a different group, similarly elements 8, 9, A, B are in the same group and C, D, E, F are in a different group, and so on. $key[6]$ ranges from 1 to $(key[0]-1) \times (key[1]-1)$.

2.2.8. *reverse* ($key[7]$)

If $key[7]$ is 1 then the elements in all the groups will be in a group wise reverse order with respect to their initial positions in the group. If this value is 0 then no such reverse operation occurs. For $key = [12, 6, 1, 1, 1, 14, 4, 0, 3, 2]$, the *TT* is shown in Figure 6. For $key = [12, 6, 1, 1, 1, 14, 4, 1, 3, 2]$ the groups get reversed and the corresponding *TT* is shown in Figure 7.

	6	5	4	3	2	1
18	3	2	1	0	7	6
17	5	4	B	A	9	8
16	F	E	D	C	J	I
15	H	G	N	M	L	K
14	R	Q	P	O	V	U
13	T	S	Z	Y	X	W
12	d	c	b	a	h	g
11	f	e	l	k	j	i
10	p	o	n	m	t	s
9	r	q	x	w	v	u
8	z	y				
7						

Figure 7: *Temporary Table (TT)*

2.2.9. *final_sum* ($key[8]$)

$key[8]$ will be explained later.

2.2.10. *power* ($key[9]$)

In the process of message level encryption, we need to calculate a unique integer corresponding to each character present in *TT*. This integer may be calculated as $(rh)^{key[9]} + (ch)^{key[9]}$. For example, for the following $key = [12, 6, 1, 1, 1, 14, 4, 1, 3, 2]$ the integer value of G is $(15)^2 + (5)^2 = 250$, similarly the integer value for 9 is $(17)^2 + (2)^2 = 293$, and so on. It is possible that two different characters may end up in the same integer, this situation is called integer collision. If integer collision occurs then we keep on increasing the integer value of the later characters in *TT* until it gets a unique value. Powering ch and rh with $key[9]$ has two benefits. 1). It makes the encryption itself more random and 2). The Probability of integer collision becomes

very less. Range of $key[9]$ from 1 to any integer depends on the capabilities of the service provider and consumer.

It is important to note that the rules outlined for the Tag Table are for demonstrating the idea and are open to modification. The main idea is to bring in as much randomness as possible so as to make it impossible for an adversary to guess the contents.

2.3. *Encryption and Decryption engines*

A client's PUT or POST request that may be in XML, JSON or plain text is first sent to the Encryption-Decryption engine. In the Encryption-Decryption engine at the client side, the request message gets converted into a string of numbers. Subsequent to this, the HTTP/S client encapsulates this string of numbers within an HTTP/S request (PUT or POST) packet and sends it to the server. At the server end, the HTTP/S server reads the string of numbers and sends it to the Encryption-Decryption engine at the server end. The Encryption-Decryption engine at this end decrypts the string of numbers into the corresponding XML and/or JSON form and finally delivers it to the Web service. The same process repeats with the response from the server to the client. The Encryption and decryption engines used here comprise the following sub-components:

2.3.1. *Temporary Table (TT)*

Temporary Tables (TT) have been discussed at length in the earlier sub-sections. We assume that the contents of this table comprise only printable ASCII characters. Each element in the table, comprising printable ASCII characters, is uniquely defined by a corresponding pair of rh and ch . The Structure of the table and the position of various characters in the cells is established by the parties in conversation using the symmetric key as discussed earlier. The *TT* is deleted after the creation of the symbol table.

2.3.2. *Symbol Table (ST)*

This comprises a table with two columns without a row or column header. The table maps each printable ASCII character to a unique integer derived from *TT* using the symmetric key. The table is mainly used for special encryption and decryption of ASCII printable RESTful content called *symbol table based encryption (STBE)* and *symbol table based decryption (STBD)*. The Creation of *ST*, its purpose, and the various steps of *STBE* and *STBD* are covered in the next section.

2.3.3. *Tag Table (TAT)*

This component includes a table with two columns without a row or column header. The table maps the *non-variable* parts of a RESTful message (XML, JSON or HTML) to a unique integer. The table is used for another encryption and decryption on ASCII printable RESTful contents called *tag table based encryption (TATBE)* and *tag table based decryption (TATBD)*. The creation of *TAT*, its purpose and the various steps involved in *TATBE* and *TATBD* are covered in the next section.

3. The Approach

Before further discussion we assume that the 10 element key ($key = [12, 6, 1, 1, 1, 14, 4, 1, 3, 2]$) has already been exchanged between the two parties and TT has been created at both ends. In the following subsections, the remaining encryption procedure is discussed based on the 10-element key and the TT created based on this key (Fig. 7). We will be using a running example for better understanding of the approach. Figure 8 summarises our approach. The figure describes both encryption and decryption.

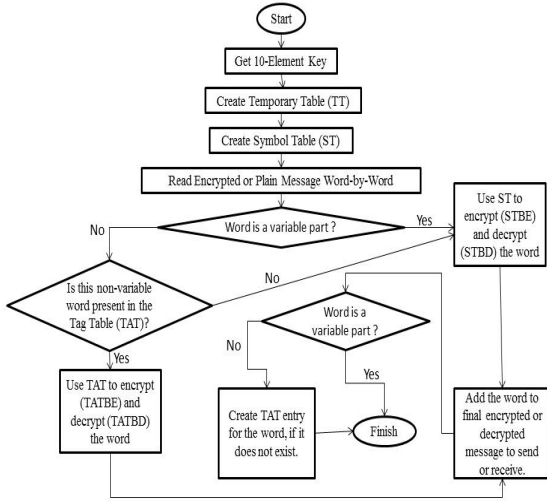


Figure 8: Flow Chart of our Approach

Our purpose is to provide a substitution cipher which converts the plain-text into a series of numbers. Such substitution cipher removes the need of XML canonicalization. As shown in Figure 8, after the creation of TT , the Symbol Table (ST) gets created as shown in Figure 9. Refer to the various *variable* and *non-variable* parts of both encrypted and plain messages as words. The symbol table maps each character in a non-header cell of the TT to a unique integer. We use this table to replace each character in the XML or JSON documents to a corresponding unique integer. For example, the word “*iiti*” gets converted into “122122104122”. This is called ST Based Encryption i.e. $STBE$. Decrypting “122122104122” back into “*iiti*” is called ST based decryption ($STBD$). The process of ST creation, $STBE$ and $STBD$ are discussed in the following sub-sections. Since the ST can be created based on the 10-element key, both parties can start encryption and decryption using ST without knowing the structure of the message. Using $STBE$ results in a large size encrypted message because each character either in the variable or non-variable parts gets converted to a unique integer. This is why we introduced the concept of Tag Table (TAT). The TAT maps the whole *variable* parts of the message to a unique integer (Fig. 10). Using the Tag Table Based Encryption ($TATBE$) results in an encrypted message size that is smaller than the $STBE$. For example, the tag “*root*” can be converted into “04”

using the $TATBE$ instead of “0117126126104” using the $STBE$. Decrypting “04” back in “*root*” is called the Tag Table Based Decryption ($TATBD$). The process of the TAT creation, $TATBE$ and $TATBD$ are discussed in the following subsections. The idea is to use the $STBE$ and $STBD$ for *variable* parts and for those *non-variable* parts of the message whose TAT entries still do not exist on either side. When the $STBE$ and $STBD$ are being used for the *non-variable* parts, the Tag Table (TAT) entry for the same must be created for later use. Use $TATBE$ for those *variable* parts whose entries are present in the TAT at both sides.

3.1. Symbol Table Creation

The Symbol Table (ST) gets created by the client and server based on the Temporary Table (TT) in the following manner: The ch and rh of all *non-header* characters inside the TT are identified and the corresponding unique number for each *non-header* symbol in the same is calculated. The procedure for creating ST is shown below:

1. For each *non-header* and *non-empty* cell in TT , the printable character is taken out and its corresponding unique integer value is calculated as $value = (rh)^{key[9]} + (ch)^{key[9]}$. For example TT in Fig. 7, character ‘j’ has $rh = 11$ and $ch = 2$. The calculated $value = (11)^2 + (2)^2$. So $value$ corresponding to ‘j’ in TT is 125.
2. The number of digits in $value$ is made the same as the value of $key[8]$. In doing this, we first calculate $diff = key[8] - (no. \text{ of digits in } value)$ and then proceed to step 3. For example if $key = [12, 6, 1, 1, 1, 14, 4, 1, 3, 2]$ ($key[8]$ is 3) then $diff = 3 - 3 = 0$.
3. We calculate the final value that is to be inserted in ST corresponding to the given *non-header* character of TT . We do it as $final_value = value \times 10^{diff}$. For example $final_value$ of ‘j’ is $125 \times 10^0 = 125$.
4. The final record to be inserted in ST is the ($character, final_value$) pair. The $final_value$ is the primary key of ST . If a record in ST exists with the same $final_value$ then go to step 5 otherwise, go to step 6. For example (‘j’, 125) is going to be inserted in ST . Before insertion of this record, we first check if a record in ST exists with the same $final_value$ (125). For now, we assume that no such record exists, and therefore go to step 6. If (‘j’, 125) is already in ST and (‘o’, 125) has to be inserted we go to step 5.
5. When a record with the same $final_value$ already exists in ST , the $final_value$ of the record is incremented by 1 and checked for uniqueness. This is continued until a unique $final_value$ is found. Subsequently, step 6 is executed. Note that even if we considered $final_value$ as a primary key more than one entry of a *character* is not possible, because each character is unique in TT . For example before inserting a record for character ‘o’, we increase its $final_value$ to 126. If a record with $final_value$ 126 also exists then we make it (‘o’, 127), otherwise, we record (‘o’, 126) and proceed to step 6.
6. Insert the record ($character, final_value$) in ST .

A subset of ST is shown in Fig.9.

r	117
o	126
t	104
a	153
1	340
v	850
l	137
u	820
e	146
2	349
n	116
m	109

Figure 9: A Subset of ST

3.2. Symbol Table Based Encryption (STBE) and TAT Creation at Server Side

By now both the client and server should have the same Temporary Table (TT) and Symbol Table (ST). The client now sends its first request to the server. The requested resource would comprise several *variable* and *non-variable* parts (either XML or JSON). The *non-variable* part could be any name and *variable* part could be of any data-type. Until this point, the client only knows the URI of the resource and does not know anything about the resource representation. If the client makes an HTTP request that does not require an HTTP body (e.g GET) then the server responds after encrypting the response message using ST as STBE. The client decodes this using the Symbol Table Based Decryption (STBD). The decrypted message gets converted into XML or JSON. In the decrypted document the name of the *non-variable* parts and data type of the *variable* parts will be clear to the client. We will explore STBD in a later sub-section.

If the client request requires an HTTP body (e.g PUT, POST) and the client does not have any idea about the resource representation in the request body (i.e what to send in the request body) then the client will send an empty POST request on the URI. In such a case, the server sends an encrypted resource representation using the STBE. The STBE is used by the server when the client is not in the know of the *non-variable* parts of the message. As soon as the client becomes aware of the *non-variable* parts, the server stops encrypting the message using the STBE and starts encrypting it using the Tag Table Based Encryption (TATBE). We will explore TATBD in a later sub-section.

In the eventuality that the server introduces a new *non-variable* part in between a conversation, the new *non-variable* part is encrypted using STBE and the encryption of the rest of the *non-variable* parts is done through TATBE. The *variable* parts of the message always get encrypted using STBE. The client never needs to encrypt the *non-variable* parts of its request body using the STBE. This is because the client can never introduce a new *non-variable* part in the request. The client, however, needs to decrypt the *non-variable* parts of the response sent out by the server that is encrypted using STBE. The process of encrypting XML1 using STBE and creation of Tag Table (TAT) for each *non-variable* parts of the message happens

simultaneously at the server side is depicted in the steps below.

Lets assume that the encrypted message gets stored in a string variable *STEnc*. The initial value of *STEnc* is *null*. We have two more variables *noOfNonVars* and *noOfDigitsForNonVars*, both are initially 0. The *variable* and *non-variable* parts of the XML are taken in the same order that they appear in the document. We together call these parts (the *variable* and *non-variable* parts of the XML) a *word*. For example in XML1 the order of occurrence of the words are “root”, “attr1”, “value1”, “attr2”, “value2”, “name”, “iti”, “/name”, “value”, “2”, “/value” and “/root”.

In the case of web service composition different tags may belong to different service providers. Different tags are therefore encrypted using different keys. The receiver must be notified about tag numbers which have been encrypted by its key. There are therefore comma separated tag numbers appended at the beginning of *STEnc* followed by space. For a tag number present at the beginning of *STEnc*, the client has access to all child elements of the given tag.

For example (1,) at the beginning of *STEnc* says that the receiver has access to the tag “root” and its children in XML1 (i.e whole document). If (2,) present at the beginning of *STEnc* says that the receiver has access to the tag “name” and its children in XML1. The presence of (2,3,) at the beginning of *STEnc* says that the receiver has access to the tags “name” and “value” and their children in XML1. Normally common tags like “root” get encrypted using an agreed upon group-key among members.

If word (1,) gets added in *STEnc*, then *STEnc* which was initially *null* becomes:
STEnc = “1, ”.

1. Repeat step-2 to step-7 for each word in the XML.
2. If the word is a tag then the same is converted into a string of numbers preceded by 0 using the ST reference for each character of the word. This string of numbers is then concatenated into a single string *STEnc* followed by a space and step 7 is executed.

For example, the first word of XML1 is a tag “root”. This consists of four characters ‘r’, ‘o’, ‘o’ and ‘t’. The ST entries corresponding to these characters are ‘117’, ‘126’, ‘126’ and ‘104’ respectively. These numbers get concatenated as “0117126126104”. Therefore, the encrypted form of the tag *root* along STBE is “0117126126104”. Concatenate the encrypted form of the word in *STEnc* followed by a space. If *STEnc* was initially “1, ” and then it gets updated as:

STEnc = “1, 0117126126104 ”.

3. If the word is an attribute-name, the same is converted into a string of numbers preceded by 00 using the ST reference for each character of the word. This string of numbers is then concatenated into *STEnc* followed by a space and step 7 is executed.

For example, the second word of XML1 “attr1” is an attribute-name. This consists of five characters ‘a’, ‘t’, ‘t’, ‘r’ and ‘1’. The Symbol Table ST entries corresponding to these characters are ‘153’, ‘104’, ‘104’, ‘117’ and ‘340’ respectively. These numbers are con-

catenated as “00153104104117340”. The encrypted form of the attribute-name *attr1* along *STBE* is therefore “00153104104117340”. Concatenate the encrypted form of the word in *STEnc* followed by a space. If *STEnc* was initially “1, 0117126126104 ”, it gets updated as: *STEnc* = “1, 0117126126104 00153104104117340 ”.

4. If the word is an attribute-value, the same is converted into a string of numbers preceded by 000 using the *ST* reference for each character of the word. This string of numbers is then concatenated into *STEnc* followed by a space and step 7 is executed.

For example, the third word of **XML1** ‘*value1*’ is an attribute-value. This consists of six characters ‘v’, ‘a’, ‘l’, ‘u’, ‘e’ and ‘1’. The *ST* entries corresponding to these characters are ‘850’, ‘153’, ‘137’, ‘820’, ‘146’ and ‘340’ respectively. These numbers get concatenated as “000850153137820146340”. Therefore, the encrypted form of the attribute-value *value1* along *STBE* becomes “000850153137820146340”. As earlier, this is concatenated with *STEnc* followed by a space. If *STEnc* was initially “1, 0117126126104 00153104104117340 ”, it gets updated as:

STEnc = “1, 0117126126104 00153104104117340 000850153137820146340 ”.

5. 0 is used to represent a closing tag. Whenever a closing tag is encountered 0 is concatenated with the string *STEnc* followed by a space.
6. If the word is a *variable* part of the XML the same is converted into a string of numbers using the *ST* reference for each character of the word. This string of numbers is then concatenated into *STEnc* followed by a space.

For example, the seventh word of **XML1** “*iiti*” is a *variable* part of the XML. This consists of four characters ‘i’, ‘i’, ‘t’ and ‘i’. The *ST* entries corresponding to these characters are ‘122’, ‘122’, ‘104’ and ‘122’ respectively. These numbers are concatenated as “122122104122”. This string of numbers is then concatenated into *STEnc* followed by a space. After completion of the encryption of the whole **XML 1**, the encrypted message *STEnc* looks like this:

STEnc = “1, 0117126126104 00153104104117340 000850153137820146340 00153104104117349 000850153137820146349 0116153109146 122122104122 1 0 0850153137820146 349 2 0 0”.

Since *key[8]* decides the size of the Symbol Table Based Encrypted message, the range of *key[8]* is from 1 to an integer depending on the network latency between the service provider and the client.

7. This step is the *TAT* creation step. This step is arrived at only for *non-variable* parts of the XML. The numbers corresponding to each character in the *non-variable* words are added and stored in the variable *sum*. The number of digits in the integers representing the *non-variable* parts of the XML (say *noOfDigitsForNonVars*), should be minimum and such that it can accommodate all *non-variable* parts in it. It can be decided by the server and client separately and automatically based on the number of *non-variable*

parts currently in use for communication (say *noOfNonVars*). The following calculations are next done:

- (a) *noOfNonVars* can be calculated as a summation of the number of new *non-variable* parts introduced in the current message and the number of entries already present in the *TAT*.
- (b) *noOfDigitsForNonVars* = $\lceil \log_{10}(\text{noOfNonVars} + 1) \rceil$
Here we add 1, because *noOfNonVars* as a multiple of 10 gives a wrong impression about the *noOfDigitsForNonVars*, if *noOfDigitsForNonVars* = $\lceil \log_{10}(\text{noOfNonVars}) \rceil$. This is because we are using zero for indicating the type of the word, therefore we should not include zero in the *non-variable* part of the number mapping. If *noOfNonVars* is 10 then $\lceil \log_{10}(\text{noOfNonVars}) \rceil$ results in 1. This implies that in such a case we need to necessarily use all digits from 0 to 9 to map the various *non-variable* parts in *TAT*. Adding 1 in the above formula removes this limitation.
- (c) *diff* = *noOfDigitsForNonVars* - (no. of digits in *sum*)
- (d) *sum* = *sum* × 10^{diff}
- (e) Finally, the record (*word*, *sum*) is stored in *TAT*. However if the corresponding *sum* is already present in *TAT* then repeat: *sum* = (*sum* + 1) % $10^{\text{noOfDigitsForNonVars}}$ until *sum* gets a unique value other than 0.

For example, the tag “*root*” consists of four characters ‘r’, ‘o’, ‘o’, ‘t’. The *ST* entries corresponding to these characters are ‘117’, ‘126’, ‘126’ and ‘104’ respectively. The steps followed for the word “*root*” are:

- (a) *sum* = 117 + 126 + 126 + 104 = 473.
- (b) *noOfNonVars* = number of new *non-variable* parts introduced in the current message (7) + number of entries already present in the *TAT* (0) = 7.
- (c) *noOfDigitsForNonVars* = $\lceil \log_{10}(7 + 1) \rceil = 1$.
- (d) *diff* = 1 - 3 = -2.
- (e) *sum* = $473 \times 10^{-2} = 4$.
- (f) The record (*root*, 4) is stored in *TAT*, because no other *sum* is present in *TAT* with a value of 4. If another entry was indeed present in *TAT* with *sum* = 4, then the entry corresponding to “*root*” stored in *TAT* would have been (“*root*”, 5). The Tag Table (*TAT*) is created simultaneously with *STBE* as shown in Figure 10. The *TAT* stores the agreed upon integers corresponding to each *non-variable* part of the XML.

root	4
name	5
value	6
attr1	8
attr2	9
value1	2
value2	3

Figure 10: Tag Table (*TAT*)

There is provision that in the future new *non-variable* parts be included in the conversation, and so *noOfDigitsForNonVars* may increase. Consequently the Tag Table (*TAT*) would also need to be updated in future to accommodate new *non-variable* parts. In general, a RESTful communication does not need more than 100 different *non-variable* parts (although the same *non-variable* parts may be repeated more than 100 times in a message). It is therefore rare for *noOfDigitsForNonVars* to be more than two.

Let us suppose that three new tags are introduced later in "XML 1" as *t1*, *t2* and *t3*, then the following calculations would need to done:

- The tag "*t1*" consists of two characters '*t*' and '*1*' with *ST* mapping of '*104*' and '*340*' respectively.
 $sum = 104 + 340 = 444$.
- noOfNonVars* = number of new *non-variable* parts introduced in the current message (3) + number of entries already present in the *TAT* (7) = 10.
- noOfDigitsForNonVars* = $\lceil \log_{10}(10 + 1) \rceil = 2$.
- $diff = 2 - 3 = -1$.
- $sum = 444 \times 10^{-1} = 44$.

The newly updated Tag Table (*TAT*) is shown in Fig. 11.

root	4
name	5
value	6
attr1	8
attr2	9
value1	2
value2	3
t1	44
t2	45
t3	46

Figure 11: Updated Tag Table

- The server sends the *STEnc* to the client within an HTTP response body in the form of plain-text.

3.3. Symbol Table Based Decryption (STBD) and TAT Creation at Client Side

Prior to receiving a response to its first request, the client does not have the Tag Table (*TAT*). The client receives the *STEnc* as plain-text as a part of the response. The client decrypts the *STEnc* and simultaneously creates the *TAT*. Each space separated sub-string of numbers in *STEnc* is called a word. "(1,)", "0117126126104", "00153104104117340" etc. are the various words in *STEnc*. There are a few global variables as well. Global variables *noOfNonVars* and *noOfDigitsForNonVars* are initially 0 and the global variable *closed* is initially 1. A global string *STDec* is defined which is initially null, and a global stack *s* is defined which is initially empty. The following steps are next followed:

- The first word conveys the tag number that has been encrypted using the receiver's key. In our example, the first word is (1,) which implies that the entire encryption

(*STEnc*) has been carried out using the client's key. The client can therefore decrypt the whole encryption.

- Step-3 through step-15 are repeated from the second word onwards in *STEnc*.
- If the word starts with a 0, it implies a tag. The preceding 0 is removed from the word.
For example, to decrypt the word 0117126126104, the preceding 0 is first removed and the word becomes 117126126104.
- If the word starts with a 00, it implies an attribute-name. The preceding 00 are removed from the word.
For example, to decrypt the word 00153104104117340, the preceding 00 are first removed and the word becomes 153104104117340.
- If the word starts with a 000, it is an attribute-value. The preceding 000 are removed from the word.
For example, to decrypt the word 000850153137820146340, the preceding 000 are removed and the word becomes 850153137820146340.
- If the word does not start with a 0, it is a *variable* part of the message.
For example, the word 122122104122 does not have a preceding 0 and hence it is a *variable* part of the message.
- The remaining word is sliced into sub-strings of *key[8]* characters each.
For example, the remaining word 117126126104 is sliced into substrings of three characters each because *key[8]* in this case is 3. Post slicing the client gets the following four sub-strings: "117", "126", "126" and "104".
- The sub-strings are next converted to integers.
- If the word is *non-variable* then all its integers are added and the result is stored in a variable called *sum*.
For the word 117126126104, $sum = 117 + 126 + 126 + 104 = 473$.

- If the word is *non-variable* then the following calculations are done:

- noOfNonVars* can be calculated as a summation of the number of new *non-variable* parts introduced in the current message and the number of entries already present in the *TAT*.
- $noOfDigitsForNonVars = \lceil \log_{10}(noOfNonVars + 1) \rceil$
- $diff = noOfDigitsForNonVars - (no. \text{ of digits in } sum)$
- $sum = sum \times 10^{diff}$

For example the following calculations are done for the word 117126126104 in *STEnc*:

- noOfNonVars* = number of new *non-variable* parts introduced in the current message (7) + number of entries already present in the *TAT* (0) = 7.
- $noOfDigitsForNonVars = \lceil \log_{10}(7 + 1) \rceil = 1$.
- $diff = 1 - 3 = -2$.
- $sum = 473 \times 10^{-2} = 4$.

noOfDigitsForNonVars gets calculated for each word, and gets updated when the number of *non-variable* words gets increased. In general a RESTful communication does not need more than 100 different *non-variable* parts (although

same *non-variable* parts may be repeated more than 100 times in a message). It is rare therefore, for *noOfDigits-ForNonVars* to be more than *two*. This type of example was also discussed in the last sub-section.

11. The corresponding character for each number found after slicing the word is identified in *ST*.

In the word 117126126104 for example, “117” belongs to “r”, “126” belongs to “o” and “104” belongs to “t”.

12. These characters are concatenated in the same order that the corresponding integers appeared in the encrypted word. The concatenated string is stored in the variable *var*.

For example, in the encrypted word 117126126104 the characters “r”, “o”, “o” and “t” are concatenated to make it “root”. *var* = “root”

13. If *var* is a *non-variable* part then the following is done:

- (a) The record (*var*, *sum*) is stored in the *TAT*. However if the corresponding *sum* is already present in *TAT* then repeat: $sum = (sum + 1) \% 10^{noOfDigitForNonVars}$ until the *sum* gets a unique value other than 0.

For example the record (*root*, 4) is stored in *TAT*, because no other *sum* is present in *TAT* with a value of 4. If an entry was present in *TAT* with *sum* = 4, then the entry corresponding to “root” that would be stored in *TAT* would be (“root”, 5). The Tag Table (*TAT*) is created simultaneously with *STBD*, which is the same as the server as shown in Figure 10.

- (b) If the variable *var* is a tag and the variable *closed* = 0, then *STDec* = *STDec* +>+<+*var*. If *closed* = 1 then *STDec* = *STDec* +<+*var*. *var* is pushed into the stack *s*. The variable *closed* is updated to *closed* = 0. For *var* = “root”, *STDec* = “< root”. “root” is pushed into the stack and the variable *closed* is updated to *closed* = 0.

- (c) If the *var* is an attribute-name then
STDec = *STDec* + [space] + *var* + “=” + “”.
After making *var* = “attr1”, *STDec* = “<root attr1= ”.

- (d) If the *var* is an attribute-value then *STDec* = *STDec* + *var* + “”.
After making *var* = “value1”, *STDec* = “<root attr1= ‘value1’ ”.

14. If *var* is a *variable* part then the steps below are followed:

- (a) If the variable *closed* = 0 then *STDec* = *STDec* +>+*var*. If *closed* = 1 then *STDec* = *STDec* + *var*. The variable *closed* is updated to *closed* = 1.

- (b) Based on the data type of the decrypted *variable* part, the client decides on the data type contained by its container tag. For example the *variable* part “iiti” indicates that the container tag <name> contains a string data type.

15. If the word comprises a single character 0, a ‘pop’ operation is done on the stack *s* to find the innermost opened tag. If *closed* = 0 then *STDec* = *STDec* +>+</+pop(*s*)>. If *closed* = 1 then *STDec* = *STDec* +</+pop(*s*)>. Update *closed* = 1.

Subsequent to working on all the words of *STEnc*, the client gets *STDec* as:

```
STDec = "<root attr1='value1'
attr2='value2'><name>iiti</name><value>2</value>
</root>".
```

The created *TAT* is the same as that of the server and is shown in Figure 10.

3.4. Tag Table Based Encryprion (TATBE)

The *TATBE* exists at both the client and server sides. After the creation of the Tag Table (*TAT*) both the client and server use it to encrypt further communications. In between a conversation, if the server introduces a new *non-variable* part then the new *non-variable* part gets encrypted using Symbol Table Based Encryption (*STBE*) and the rest of the *non-variable* parts gets encrypted using *TATBE*. A client never needs to introduce a new *non-variable* part of the message. The *variable* parts of the message always gets encrypted using *STBE*.

TATBE is very similar to *STBE* with the only difference being that the Symbol Table (*ST*) is used for character-by-character encryption of the message, and here we use the Tag Table (*TAT*) to map the *non-variable* parts with unique numbers.

Lets assume that the encrypted message gets stored in a string type global variable *TATEnc*. The initial value of *TATEnc* is null. The *non-variable* and *variable* parts of the XML are considered in the same order in which they appear in the XML. The *non-variable* and *variable* parts of the XML together constitute a *word*. For example, in **XML1** the order of occurrence of the words is “<root>”, “attr1”, “value1”, “attr2”, “value2”, “<name>”, “iiti”, “</name>”, “<value>”, “2”, “</value>” and “</root>”.

Just like *STBE* there is a comma separated list of tag numbers appended at the beginning of *TATEnc* followed by a space. This list of tag numbers is an indication to the receiver on which tag and its children have been encrypted using the client’s key. For example, the number (1,) at the beginning of *TATEnc* implies that the receiver has access to the tag “root” and its children in **XML1** (i.e the whole document). The number (2,) at the beginning of *TATEnc* implies that the receiver has access to the tag “name” and its children in **XML1**. The presence of (2,3,) at the beginning of *TATEnc* indicates that the receiver has access to the tags “name” and “value” and their children in **XML1**. We assume here that the whole XML has been encrypted using the client’s key. *TATEnc* is updated to: *TATEnc* = *TATEnc* + “1,”. The steps involved in the *TATBE* are as follows.

1. Step-2 through step-7 are repeated for each word in the XML.
 2. If the word is a tag, the *TAT* is searched for this word. If the word is found in the *TAT*, its corresponding integer is fetched. The fetched integer is converted into a string and is appended with a 0. We store this string of numbers in variable the *var*. *TATEnc* is next updated to: *TATEnc* = *TATEnc* + [space] + *var*. If the tag is not available in *TAT* then go to step-6.
- For example, the first word of **XML1** is a tag “root”. This tag is found in *TAT*. The corresponding integer of the tag,

4, is fetched from the *TAT*. This integer is converted to a string and appending with a 0. $var = 04$. *TATEnc* is updated to: $TATEnc = TATEnc + [space] + "04"$.

3. If the word is an attribute-name, the *TAT* is searched for this word. If the word is found in the *TAT*, its corresponding integer is fetched. The fetched integer is converted into a string and is appended with with a 00. This string of number is stored in the variable *var*. *TATEnc* is next updated as: $TATEnc = TATEnc + [space] + var$. If the attribute-name is not available in *TAT* then goto step-6.
For example, the second word of **XML1** is the attribute-name "*attr1*". This attribute-name is available in *TAT*. The corresponding integer of the attribute-name, 8, is fetched from the *TAT*. This integer is converted to a string and appended with 00. $var = 008$. *TATEnc* is updated to: $TATEnc = TATEnc + [space] + "008"$.
4. If the word is an attribute-value, the *TAT* is searched for this word. If the word is found in the *TAT*, its corresponding integer is fetched. The fetched integer is converted into a string and is appended with a 000. This string of numbers is referred to as *var*. *TATEnc* is next updated as: $TATEnc = TATEnc + [space] + var$. If the attribute-name is not available in *TAT* then goto step-6.
For example, the second word of **XML1** is the attribute-value "*value1*". This attribute-name is available in *TAT*. The corresponding integer of the attribute-name, 2, is fetched from the *TAT*. This integer is converted to a string and appending with 000. $var = 0002$. *TATEnc* is updated to: $TATEnc = TATEnc + [space] + "0002"$.
5. 0 is used to represent the closing tag. Whenever a closing tag occurs, 0 is concatenated with the string *TATEnc* separated by a space.
6. If the word constitutes the *non-variable* part of the XML which is not found in *TAT*, the word is converted into a string of numbers using *STBE*. *TATEnc* is updated as $TATEnc = TATEnc + [space] + \text{result of STBE for word}$.
7. *STBE* is explored for the given *variable* word and *TATEnc* is updated as: $TATEnc = TATEnc + [space] + \text{result of STBE for the word}$.

On completion of the encryption of **XML 1** the encrypted message *TATEnc* looks like $TATEnc = "1, 04 008 0002 009 0003 05 122122104122 0 06 349 0 0"$.

As long as an entry for a word is present in *TAT*, the server does not use *STBE* to encrypt the word. If, however, a new *non-variable* part is introduced in the XML as shown in **XML 2**, the newly introduced *non-variable* part would need to be encrypted using *STBE* while the rest of the *non-variable* parts would be encrypted using *TATBE*. For the XML given in **XML 2** the final encrypted message becomes: $TATEnc = "1, 04 008 0002 009 0003 05 122122104122 0 06 349 0 0116850 153340 0 0"$.

XML 2:

```
<root attr1="value1" attr2="value2">
<name>iiti</name>
```

```
<value>2</value>
<nv>a1</nv>
</root>
```

3.5. Tag Table Based Decryption (TATBD)

TATBD stands for *TAT* based decryption and this exists at both the client and server ends. The Client gets *TATEnc* as plain-text in the response whereas the server gets it within the request body. The receiver decrypts the *TATEnc*. Each space separated sub-string of numbers in the *TATEnc* is called a word. (1,), 04, 008 etc. are the various words in *TATEnc*. We define a global variable called *closed* that is initially 1, a globally defined string *TATDec* that is initially *null*, and a global stack *s* that is initially empty.

1. The first word indicates the tag numbers that have been encrypted using the receiver's key. In our running example the first word is (1,). This indicates that the whole encryption (*TATEnc*) has been done using the client's key, and therefore the client can decrypt the whole XML.
2. Step-3 through 12 are repeated for the second word onwards in *TATEnc*.
3. If the word starts with a 0, it implies that it is a tag. The preceding 0 is removed from the word and it is converted to an integer.
For example to decrypt the word 04, the preceding 0 is first removed and it becomes 4. The same is then converted to an integer.
4. If the word starts with 00, it implies that it is an attribute-name. The preceding 00 is removed from the word and the same is converted to an integer.
For example to decrypt the word 008, the preceding 00 is first removed and it becomes 8. The same is then converted to an integer.
5. If the word starts with 000, it implies that it is an attribute-value. The preceding 000 is removed from the word and the same is converted to an integer.
For example to decrypt the word 0002, the preceding 000 is removed and it becomes 2. The same is converted to an integer.
6. If the word does not have a preceding 0, it implies that it is the *variable* part of the message. The variable part of the message gets decrypted using the *STBD*.
7. The *non-variable* words are first searched in *TAT*. If the word is not found in *TAT* then the same is decrypted using the *STBD*. If, however, the word is found in the *TAT* then the corresponding name is fetched and stored in the variable *var*.
8. If the word is a *tag* and *closed* = 0 then $TATDec = TATDec + > + < + var$. If *closed* = 1 then $TATDec = TATDec + < + var$. *var* is pushed into the stack *s* and *closed* is updated: *closed* = 0.
For *var* = "*root*", $TATDec = "<root"$. Word "*root*" is pushed into the stack and *closed* is update *closed* = 0.
9. If the word is an attribute-name then $TATDec = TATDec + [space] + var + "=" + ""$.
After making *var* = "*attr1*", $TATDec = "<root attr1="$.

10. If the word is an attribute-value then $TATDec = TATDec + var + ""$.
After making $var = "value1"$, $TATDec = "<root attr1='value1'"$.
11. If the word is a *variable* part of the XML and $closed = 0$ then $TATDec = TATDec + > + var$. If $closed = 1$ then $TATDec = TATDec + var$. Variable $closed$ is update to $closed = 1$.
12. If a single character 0 is found as a word then a pop operation is done on the stack s to find innermost opened tag. If $closed = 0$ then $TATDec = TATDec + > + </ + pop(s) + >$. If $closed = 1$ then $TATDec = TATDec + </ + pop(s) + >$. $closed$ is updated: $closed = 1$.

Subsequent to working on all the words of $TATEnc$ the client gets $TATDec$ as "`<root attr1='value1' attr2='value2'><name>iiti</name><value>2</value></root>`".

3.6. Communication in web service Composition Scenario

In a web service composition scenario, there is multiple service providers that work on the same message but need to have access to only some parts of the message. To realize this, the proposed approach encrypts different parts of the message with different keys. Lets consider a scenario where there is one main server (Service Provider) S and two other service providers SP1 and SP2. They commonly agree on a *key* called *group_key*.

A client C sends a request to the main server S. S needs to reply to the request with XML 2. In putting together the reply, S requires the services of SP1 and SP2. SP1 is required to update the value between the `<name>` tag and SP2 is required to update the values between the `<value>` and `<nv>` tags. SP1 and SP2 must not be able to access or update any other tags that do not belong to them. In this situation, there are three symmetric keys in use. The first one is the *key* between S and SP1 (say K1), the second one is the *key* between S and SP2 (say K2) and the third one is the agreed upon *group key* between all S, SP1 and SP2 (Say K3). The common tags of the XML will be encrypted using the *group_key* K3. Tags that are only supposed to be updated by S and SP1 must be encrypted using K1 and tags that are only supposed to be updated by S and SP2 must be encrypted using K2.

The first word of the encrypted message tells SP1 and SP2 about one or more tag numbers separated by a comma that they are supposed to process. If secret key of SP1-S is [12, 6, 1, 1, 1, 14, 4, 1, 3, 2], secret key of SP2-S is [6, 12, 1, 0, 1, 14, 3, 1, 3, 2] and *group_key* is [7, 10, 0, 0, 1, 14, 3, 0, 3, 2]. SP1 is supposed to process the second tag i.e *name* and its child. SP2 is supposed to process the third and fourth tags i.e *value* and *nv* and their children. *TAT* based Message from S to SP1 is: "2, 01 009 0002 003 0004 05 122122104122 0 07 313 0 08 356290 0 0". *ST* based Message from S to SP1 is: "2, 0232325325180 00137180180232257 000136137126157314257 00137180180232226 000136137126157314226 0116153109146 122122104122 0 0291356265326320 313 0 0410291 356290 0 0". *TAT* based Message from S to SP2 is: "3,4, 01 009 0002 003 0004 05

122122104122 0 07 313 0 08 356290 0 0". *ST* based Message from S to SP2 is: "3,4, 0232325325180 00137180180232257 000136137126157314257 00137180180232226 000136137126157314226 0116153109146 122122104122 0 0291356265326320 313 0 0410291 356265 0 0".

This web service composition scenario is just an example. Various other compositions are also possible.

3.7. Message Authentication in Web Service Composition

Message authentication is a compulsory step in any conversation where there is a possibility of updates by others in the middle because encrypted message can also be changed. For message authentication, we use existing algorithms like MD5, SHA1 etc. To exemplify this, we know from the earlier example that the tag that belongs to S and SP1 is `<name>`. As both S and SP1 want a confirmation that tag `<name>` which is the second tag is not changed by an intermediary. Therefore, to achieve message authentication after encrypting the tag `<name>`, the sender (S) prefixes K1 to the encrypted tag and creates a hash (say MD5) of it. The overall hash is attached at the end of the corresponding tag.

The sequence of steps followed by the sender (S) and receiver (SP1) is:

1. Sender encrypts the message using *TATBE* as above.
2. The *TAT* based encryption of the `<name>` tag is "05 122122104122 0". This is appended with the private key K1 by the sender as "[12,6,1,1,1,14,4,1,3,2]05 122122104122 0".
3. The sender calculates the MD5 of "[12,6,1,1,1,14,4,1,3,2]05 122122104122 0" as "adc1aeffe1fe867740f976fd55c0c481" (say D1).
4. The *TAT* based encryption of the `<root>` tag is "01 009 0002 003 0004 05 122122104122 0 07 313 0 08 356290 0 0". This is appended with the group key K3 by the sender as "[7,10,0,0,1,14,3,0,3,2]01 009 0002 003 0004 05 122122104122 0 07 313 0 08 356290 0 0".
5. The sender (S) calculates the MD5 of "[7,10,0,0,1,14,3,0,3,2]01 009 0002 003 0004 05 122122104122 0 07 313 0 08 356290 0 0" as "72afa9838090da9c5d82d2060c42f48c" (say D2).
6. Before sending the reply the sender appends these digests just after closing the corresponding tag for which digests have been calculated. D1 is appended after closing the second tag and D2 is appended after closing of the "`<root>`" tag.
7. The sender sends the message: "2, 01 009 0002 003 0004 05 122122104122 0 D1 07 313 0 08 356290 0 0 D2". There is no need to authenticate the third and fourth tags because these do not belong to SP1.
8. The first word "(2,)" conveys to the receiver that it has to work on the second tag. It is known to all that the first tag "`<root>`" is encrypted using a group key, as this tag contains all other tags.
9. The *variable* part at the end of a tag that contains alphabetic characters is the message digest of the corresponding

tag. For example, D1 is present just after closing the second tag, which is why the receiver considers it as a digest of the second tag and its children. Similarly, D2 is the digest of the first tag (“<root>”) and its children. Although the whole “<root>” tag and its children are not encrypted using the client’s key, the overall digest D2 is created by appending the K3.

10. The receiver takes out the second word of the encrypted message and D1, and further calculates its digest in the same way as the sender did in step-3 using K1. If the digest calculated by the receiver is the same as D1 then the message is authentic. If they are not the same, the message will be rejected.
11. The receiver calculates the digest of the whole message that the sender did it in step-5 using K3. If the digest calculated by the Receiver is the same as D2 then the message is authentic. If they are not same, the message will be rejected.

A Similar process is followed to authenticate the Symbol Table Based Encryption. In case of communication between S and SP2 the sender will use K2 instead of K1. The message from S to SP2 is: “2, 01 009 0002 003 0004 05 122122104122 0 07 313 0 5f4ffdd89acc919420ac885e6017bcfc 08 356290 0 44e9e15af5f4289bab86c90e0d9398d1 0 72afa9838090da9c5d82d2060c42f48c”.

3.8. Working With JSON Data

JSON has a different structure from XML but both are interchangeable with each other. We can easily convert XML 1 to JSON 1 as shown.

We treat JSON names, values of a JSON name that start with a ‘-’ as *non-variable* words and double quoted JSON names as *variable* words. An array in JSON is considered a repetition of a tag and the values in between them are consecutive elements of an array. Closing the curly bracket denotes the end of the innermost open tag.

As XML and JSON are equivalent, their encryption as string of numbers must also be the same.

4. Advantage of the proposed approach

Here we present the salient feature of the proposed technique highlighting its strength as compared to existing techniques.

4.1. Need for XML canonicalization is eliminated

As mentioned earlier, canonicalization converts logically equivalent XML documents to ones with identical physical structures. We require XML canonicalization if we encrypt an XML or part of the XML byte-by-byte. Formal security techniques provided by w3c and OASIS for SOAP based web services [3] need XML canonicalization. But since our encryption technique is a text based substitution (Character to number), we do not need XML canonicalization.

4.2. Need for content negotiation is eliminated

As the encryption of XML and JSON that represent the same data results in identical messages in the proposed approach. The message may also be decrypted to any of the two forms. The approach therefore seamlessly works with both XML and JSON contents.

4.3. No need to send extra HTTP headers

We do not send any extra HTTP headers as in the case of Serme, G. et. al [11]. In our case the agreement on a key between a service provider and its client results in a compromise of statelessness of RESTful web services to a very small extent.

4.4. Resource representation in encrypted form

In the dynamic web services composition where a service provider may enter or leave the group randomly, the resource itself becomes a sensitive document. Putting the resource representation in the public domain is not good for such a scenario. As already discussed in our approach, the client requests a resource whose representation is not known to it with a GET method and the encrypted resource is transferred using Symbol Table Based Encryption (STBE). If the client uses an empty POST or PUT the request the server explicitly sends the encrypted resource representation to the client using STBE.

4.5. Suitable for RESTful web services composition

The proposed approach facilitates different parts of the same message to be encrypted with different keys. The various intermediaries are notified about the tag number on which they have to work. An intermediary therefore does not get any sense of others’ data. If an intermediary does try to change others’ data, the receiver will simply reject it because of message authentication.

4.6. Size of encrypted message

The size of an HTTP message should be as small as possible in RESTful communication. The encrypted message created using this approach is much smaller in size than that of other existing algorithms. Further, if the number of *non-variable* characters is larger than the number of *variable* characters, the size of the encrypted message is even smaller than the size of the actual message itself. However if the number of *variable* characters dominates over the number of *non-variable* (which is relatively rare) characters the size of the encrypted message using our approach may sometimes become larger than that of existing approaches. In this case the size of the encrypted message largely depends on *key*[8]. Table 1 describes the size comparison of encrypted messages using various encryption algorithms on various types of XML data. Here we do encryption on the whole XML, not on a given part of the XML. The key used for STBE and TATBE is [12,6,1,1,1,14,4,1,3,2]. Here we include various characters like <, >, /, “, = etc used for the *non-variable* parts in the count of the number of characters used in the *non-variable* parts.

Table 1: Size comparison of various encryption techniques on various XML

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8	Column 9	Column 10
12	12	161	6	101	318	473	441	565	376
18	45	637	24	404	1209	1665	1625	2185	1480
22	89	1235	48	808	2397	3245	3213	4345	2938
18	45	627	24	627	1432	1965	1921	2845	2140
8	27694	341167	27644	64961	433776	615258	578380	637558	360770
1	1	39	1	1	41	64	40	59	9
3	3	12	1	12	26	89	57	57	51
1	1	7	1	17	25	32	40	59	57

Column1 represents the number of unique {non-variable} parts of the document.

Column2 represents the number of {non-variable} parts used in the document.

Column3 represents the number of characters in the {non-variable} parts.

Column4 represents the number of {variable} parts.

Column5 represents the number of characters in the {variable} parts.

Column6 represents the total number of characters in the XML (Including new line characters and a few space characters).

Column7 represents the number of characters in AES (Rijndael 256) encryption.

Column8 represents the number of characters in 3DES encryption.

Column9 represents the number of characters in STBE encryption.

Column10 represents Number of characters in TATBE encryption.

4.7. Hard to attack

This is especially true in the condition that a service provider within the group is the attacker. This, in fact, makes this approach especially useful. The following points make this encryption hard to attack:

1. The *10-element* key is generated randomly and sent to the other party using TLS. It is therefore not possible for other service providers to sniff the key.
2. A brute force attack is not possible. This is so because first, it is very hard to go through all possible values for all ten elements, and then to create various tables for all possibilities, and further to decrypt a part of the XML that does not belong to the attacker with all possible keys. Even if the attacker manages the above, it will not be sure which one is the correct decryption.
3. Message authentication prevents an attacker from changing the unauthorized part of the XML through hit and trial.
4. The *Known plain-text attack* [12] is the most dangerous for this approach. This attack is possible if the attacker has a valid plain-text/cipher-text pair. Using this, the attacker can try to guess the further encrypted message or even the key. If somehow the attacker is able to map between the various characters and the number or the tag-name and the number, then all of the encryption will be compromised. The question here is how will the attacker get a valid cipher-text/plain-text pair in a web service composition scenario.
5. The *chosen plain-text attack* [13] is an attack that presumes that the attacker can get a cipher-text for any random plain-text and can try to guess the further conversation or the key itself. In case of web service composition the attacker is a service provider and would have a different shared key, and therefore it can not possibly get an encrypted message with other's key.

6. Various security aspects on RESTful web services are described by OWASP [14]. Here we are dealing with only the message level security. In our approach XML and JSON input validation and message integrity are provided through message authentication of the XML or JSON document.

5. Conclusion

In this paper, we proposed a novel approach for message level security in RESTful web services. This approach is a kind of substitution cipher which replaces different characters by a unique integer. Various tag-names, arguments and values of arguments were also replaced by corresponding unique integers. Substitution from character to number removes the need for XML canonicalization. This approach removes the need for content negotiation for the same resource among service providers and clients, and also reduces the size of the overall encrypted message. We applied this approach on several XML and JSON data and found that for most of the cases our approach resulted in smaller sized encrypted messages than those of existing approaches. We also demonstrated the efficiency of the proposed message in web services composition scenarios through encrypting different tags with different keys, and also maintained message authentication. The discussed algorithms were implemented in JAVA.

Acknowledgment

The authors would like to thank the "Department of Electronics and IT (DeitY), Government of India" for funding this project.

References

- [1] R. T. Fielding, “*architectural styles and the design of network-based software architectures*,” Ph.D. dissertation, University of California, Irvine, 2000.
- [2] Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [3] Nils Agne Nordbotten, “*XML and Web Services Security Standards*”, IEEE Communication Surveys & Tutorials, VOL. 11, NO. 3, Third Quarter 2009.
- [4] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia, Ed Simon, “*XML Signature Syntax and Processing Version 2.0*”, <http://www.w3.org/TR/xmlsig-core2/>, W3C Working Group Note 23 July 2015.
- [5] Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, Magnus Nyström, “*XML Encryption Syntax and Processing Version 1.1*”, <http://www.w3.org/TR/xmlenc-core1/>, W3C Recommendation 11 April 2013.
- [6] Chown, P., “Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)”, RFC 3268, DOI 10.17487/RFC3268, June 2002, <<http://www.rfc-editor.org/info/rfc3268>>.
- [7] Kelly, S., “Security Implications of Using the Data Encryption Standard (DES)”, RFC 4772, DOI 10.17487/RFC4772, December 2006, <<http://www.rfc-editor.org/info/rfc4772>>.
- [8] Eastlake 3rd, D. and T. Hansen, “US Secure Hash Algorithms (SHA and HMAC-SHA)”, RFC 4634, DOI 10.17487/RFC4634, July 2006, <<http://www.rfc-editor.org/info/rfc4634>>.
- [9] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”, RFC 4492, DOI 10.17487/RFC4492, May 2006, <<http://www.rfc-editor.org/info/rfc4492>>.
- [10] John Boyer, Glenn Marcy, Pratik Datta, Frederick Hirsch, “Canonical XML Version 2.0”, W3C Working Group Note 11 April 2013.
- [11] Serme, G., de Oliveira, A.S., Massiera, J., Roudier, Y., “*Enabling Message Security for RESTful Services*”, 2012 IEEE 19th International Conference on Web Services.
- [12] Alex Biryukov, “*Known Plaintext Attack*”, pp. 704-705, Springer US, Boston, MA, 2011.
- [13] Alex Biryukov, “*Chosen plaintext attack*”, pp. 205-206, Springer US, Boston, MA, 2011.
- [14] “REST Security Cheat Sheet”, OWASP Cheat Sheets, 10 April 2016, <https://www.owasp.org/index.php/REST_Security_Cheat_Sheet>.